ECM2418 Computer Languages and Representations
Continuous Assessment 2: Prolog

Dr Enrico Malizia

Handed out	Handed in
Monday 30th October 2017 (T1:06)	Wednesday 29th November 2017 (midday) (T1:10)

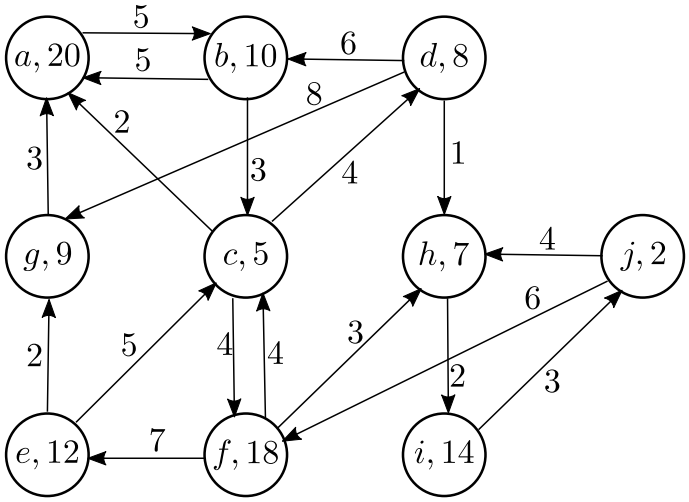
This Continuous Assessment is worth 15% of the module mark.

Your submission for this CA will be both on paper, using BART, and electronic, using the EMPS Anonymous Electronic Coursework Submission system (`http://empslocal.ex.ac.uk/cgi-bin/submit/prepare`). In this system, you should submit to the folder entitled "2017-11-29 ~ Enrico Malizia ~ CA2 Prolog". *Remember that the submission deadline, for both the paper and the electronic version, is on* ***midday*** *of the hand in day.*

All students are reminded of the University regulations on academic honesty and plagiarism.

Your task

For this coursework, your task is to write a Prolog program to solve a number of problems on (double-weighted directed) graphs. In particular, we assume that we are modeling the road network of a city in a Prolog database. In this database, a fact `vertex(a)` (which is a vertex of the graph) states that in the city there is a neighborhood called a, while a fact `edge(a,b,5)` (which is a weighted edge of the graph) states that there is a road between neighborhood a and neighborhood b, that the road can be travelled *from a to b*, and that the travel time of the road is 5 minutes. The travel time of a road is assumed to be an integer. If the road between a and b can be travelled in both directions, then in the database there are the two facts `edge(a,b,5)` and `edge(b,a,5)`. From ELE you can download a Prolog file `graph_db.pl` containing the Prolog representation, according to the above mentioned scheme, of the following graph. For now, ignore the weights inside the vertices, as well as the facts `costFS/2` in the database encoding these.



(1) Define a Prolog function `checkGraph` to check whether the database in input is consistent. Here we assume that we need to check only the following two types of errors:

- **Vertex consistency:** roads listed in the database have to connect neighborhoods that are actually listed in the database. For example, if in the database there is the fact `edge(a,b,5)`, then facts `vertex(a)` and `vertex(b)` must be present in the database as well.
- **Weight consistency:** the time to travel a road must be at least 1 minute; and if a road can be travelled in both directions, then the time to travel the road must be the same in the two directions. For example, a fact `edge(a,b,-1)` is illegal, and the pair of facts `edge(a,b,2)` and `edge(b,a,3)` is illegal.

The function `checkGraph` should print information regarding the errors present in the database. For example, consider the following database:

```
vertex(a).
edge(l,k,-1).
edge(k,l,4).
edge(a,z,-2).
```

The query '`?- checkGraph.`' prints the following information:

```
endpoint l of edge (l,k) is not a valid vertex
endpoint k of edge (l,k) is not a valid vertex
endpoint k of edge (k,l) is not a valid vertex
endpoint l of edge (k,l) is not a valid vertex
endpoint z of edge (a,z) is not a valid vertex
edge (l,k) has weight -1
edge (l,k) has weight -1 and edge (k,l) has weight 4
edge (k,l) has weight 4 and edge (l,k) has weight -1
edge (a,z) has weight -2
```

Observe that duplication of printed information is kept to a minimum. For example, "`edge (l,k) has weight -1`" is printed only once. For simplicity, we assume that edges (l, k) and (k, l) are different. Therefore, we write twice the information relative to the non-consistency of their endpoints; and we print twice the information that two edges have different weights (see in the example the printed information regarding the endpoints and the weights of edges (l, k) and (k, l)). **(20 marks)**

(2) Let us now define some Prolog (utility) functions that will be useful for the subsequent parts of the coursework. In particular, define the following Prolog functions:

- `member(X,L)`: verifies whether X is a member of the list L;
- `isSet(L)`: for the list L, verifies whether L is a set, i.e., there are no duplicates in L;
- `lastElement(Z,L)`: verifies whether Z is the last element of list L;
- `append(L1,L2,L)`: computes list L that is the result of appending list $L2$ at the end of list $L1$;
- `intersect(A,B,C)`: computes list C that is the set obtained as the intersection of sets A and B (represented as lists).

(5 marks)

(3) In this part, the task is to write functions to compute paths between vertices in the graph. Computing a path in a graph could be a bit challenging, because, given the particular resolution strategy adopted by Prolog, if the graph is cyclic, then a non-properly designed function could not terminate. In order to avoid endless loops, a Prolog function looking for paths in (cyclic) graphs has to keep track of the

vertices already visited (to avoid exploring areas of the graph that have been already visited). Consider the following Prolog function `path(X,Y)` that verifies whether there is a path *from* vertex X *to* vertex Y. The following function correctly works also over cyclic graphs.

```
path(X,Y):- pathHelper(X,Y,[]).
pathHelper(X,X,_).
pathHelper(X,Y,VISITED):- edge(X,Z,_),\+member(Z,VISITED),pathHelper(Z,Y,[X|VISITED]).
```

Starting from the code above, define the following Prolog functions (define first the most general functions; the other functions exploit the more general ones): **(25 marks in total)**

- `wPathRoute(X,Y,L,W)`: computes L and W, where L is (a list of vertices that is) a route going from X to Y, and W is the total travel time for L. For example, for the graph depicted above, answers to the query '`?- wPathRoute(a,h,L,W).`' are:

```
L = [a, b, c, d, h],
W = 13 ;
L = [a, b, c, f, h],
W = 15 ;
false.
```

(7 marks)

- `pathRoute(X,Y,L)`: computes a route L going from X to Y; **(1 mark)**
- `wPath(X,Y,W)`: computes/verifies whether there is a path from X to Y whose total travel time is W; **(1 mark)**
- `path(X,Y)`: verifies whether there is a path from X to Y; **(1 mark)**
- `wPathAvoidSetRoute(X,Y,SET,L,W)`: computes L and W, where L is a route going from X to Y avoiding all the vertices in the set SET, and W is the total travel time of L; **(3 mark)**
- `pathAvoidSetRoute(X,Y,SET,L)`: computes a route L going from X to Y avoiding all the vertices in the set SET; **(1 mark)**
- `shortestPathAvoidSet(X,Y,SET,L,W)`: computes L and W, where L is a route, with minimum total travel time (which is returned in W), going from X to Y, and avoiding all the vertices in the set SET; **(5 marks)**
- `shortestPath(X,Y,L,W)`: computes L and W, where L is a route, with minimum total travel time (which is returned in W), going from X to Y; **(1 mark)**
- `connectedGraph`: returns `true` if and only if the graph is connected (i.e., there is a path between any two distinct vertices). **(5 mark)**

(4) A sightseeing tour of a city is a route within the city that passes through *all* the neighborhoods of the city exactly *once*, and the tour ends in the same neighborhood where it started (in the literature this is called a Hamiltonian cycle). For the graph depicted above, a sightseeing tour is $(a, b, c, d, h, i, j, f, e, g, a)$. Define the following Prolog functions:

- `buildSST(T)`: computes a sightseeing tour of the city and the result is returned in T. For example, for the graph above, part of the answer to the query '`?- buildSST(T).`' is:

```
T = [b, c, d, h, i, j, f, e, g, a, b] ;
T = [c, d, h, i, j, f, e, g, a, b, c] ;
T = [d, h, i, j, f, e, g, a, b, c, d] .
```

(not all answers are shown). (Hint: start by thinking how computing the permutations of the vertices would help to solve the problem).

- `buildSSTWithStart(V,T)`: computes a sightseeing tour of the city starting from neighborhood V and the result is returned in T. For example, for the graph above, the answer to the query '`?- buildSSTWithStart(a,T).`' is:

```
T = [a, b, c, d, h, i, j, f, e, g, a] ;
false.
```

which means that, apart from `[a, b, c, d, h, i, j, f, e, g, a]`, there are no other (different) sightseeing tours departing from a.

(25 marks)

(5) We want to decide where to build fire stations in the city in order for all the city to be safe. The whole city is safe if, for every neighborhood X, either a fire station is located in X, or X can be reached from a fire station within 5 minutes. However, the city council does not have an unbounded budget available for this, and building fire stations in different neighborhoods has different costs. In the graph depicted above, the weights inside the vertices are the costs to build fire stations in those neighborhoods. For example, building a fire station in neighborhood a costs £20m, and the fact `costFS(a,20)` in the database states precisely this. The cost of a fire station is assumed to be an integer number. We assume that this part of the database is always valid, i.e., there are no errors in the database (like, negative costs, neighborhood for which the cost is not stated, etc.). Define the following Prolog functions:

- `buildSafeSetFSWithinBudget(B,S)`: computes a set S, if exists, of locations where to build the fire stations to keep safe all the city, within the budget B. For example, for the graph above, among the results of the query '`?- buildSetFSSafeWithinBudget(50,S).`', there is `S = [h, e, c, a]`, whose total cost is £44m, and keeps all the city safe.
- `computeMinCostSafety(BMIN)`: computes the minimum cost (returned in $BMIN$) required to build fire stations and have all the city safe. For example, for the graph above, for the query '`?- computeMinCostSafety(BMIN).`', we get the answer `BMIN = 34.`
- `buildMinCostSafeFS(BMIN,S)`: computes a set S of locations where to build the fire stations to keep safe all the city, such that the total cost is minimum, which is returned in $BMIN$. For example, for the graph above, for the query '`?- buildMinCostFS(BMIN,S).`', we get the answer

```
BMIN = 34,
S = [h, e, c, b] ;
false.
```

stating that the minimum budget to keep all the city safe is £34m, and that `[h, e, c, b]` is the only possible set of locations where to build fire station to keep all the city safe at minimum budget.

(30 marks)

What you should submit

Your submission should comprise both an electronic submission and a hard-copy submission, as follows:

1. The electronic submission should consist of one `.pl` file, containing the Prolog code for your solution, and one `.pdf` file, containing a descriptive account of how your program meets the required functionality *and* the listing of your Prolog code.
2. The hard-copy submission should be the print of the `.pdf` file above.